Static Analysis on Executable Code - A Survey

Stefan Bygde

April 11, 2011

Abstract

This is a survey paper investigating approaches to static program analysis on executable code. Analysis on executable code is important because in some cases the source code does not describe what happens during execution well enough. In other cases, only the executable may be available. Several difficulties are involved in analysing executables though, including incomplete information about the program flow, missing information about program variables, bit manipulation etc. This article surveys different suggested approaches to overcome some of these problems.

1 Introduction to Static Analysis

Static Analysis is a term used to denote a process for automatically extracting information about a computer program without executing it. Often this information is about a program's run-time behaviour. Such information is important for debugging purposes, compiler optimisations, formal verification of programs and other areas such as timing analysis. Thus, static analysis is a broad topic and it has many applications.

1.1 Data-Flow Analysis

In this paper, we will restrict static analysis to mean data-flow analysis, which is the most common area of static analysis. In data-flow analysis, a program is viewed as a graph representing the structure of a program. A data-flow analysis then analyses how information is propagated though this graph representation of the program.

However, while it would be desirable to extract *all* possible information about programs behaviour at run-time, this is in general not possible. Static analysis is performed by a computer program, which does not have more expressive power than the programs it analyses. Therefore, exact information about an analysed program is in general not possible to obtain. Therefore, any data-flow analysis obtains some sort of approximation of the program's behaviour. This approximation is typically desired to be conservative, that is, the information must be *correct* (or, *sound* as we will call it from now on), but not necessary *complete*.

In a nutshell, a data-flow analysis has to specify which properties should be extracted from a program. An example would be "does this program have computations resulting in an overflow?" [1]. Again, the answer to this question can not in general be answered exactly, however, an analysis should be able to give a correct answer by saying either "no, this program has no such computations" or "yes, this program might have such computations". The "might have" part is the approximation in this case.

When the property of interest has been specified, the analysis is usually formulated as a set of *transfer functions* which for each type of statement (where one or more statements are typically associated with a node in the program graph) describes how this property is affected by the statement. The graph together with these transfer functions then give rise to a set of equations or constraints. This system of equations (or constraints) can then be solved by various algorithms to find the *most precise* solution to this system. A good source of information about program analysis in general can be found in [27].

1.2 An Example

So, what kind of information is propagated through the program graph mentioned above? This depends of course on the analysis in question, but a common property of interest is to obtain information about the values of program variables. One common analysis derives upper and lower bounds of each variable at each point in the program. Consider the following program:

```
1: n:=5
2: i:=0
3: while i < n
4:  i:= i + 1
5: end while
```

A data-flow analysis could for example discover that, after any execution of line 4, i will have a lower bound of 1 and an upper bound of 5. This particular analysis is known as *interval* analysis (see Section 4). To discover these bounds, the analysis has to approximate the results of all computations made in a program without actually performing all the computations that would occur during execution. Note that the derived bounds are *correct* in the sense that a variable is *guaranteed* to stay within those bounds, however, due to approximations, the bounds may be unecessarily loose.

A typical data-flow analysis derives analysis results for all places in a program graph, not only one statement. Since analysis may be performed over different kind of program graphs (control flow graphs, flow charts, flow graphs etc) and different parts of statements may be referred to as a point in a program (a graph node, a graph edge, a statement, a row number etc), we shall from here on refer to a program's flow representation as a *program graph*, and any point of a program as a *program point*.

[1] While this is a yes/no question, data-flow analysis is not restricted to this kind of information

2 What You See Is Not What You Execute

As mentioned, static analysis in general has a large number of concrete applications, and a substantial amount of research related to static analysis has been done over the years. Most of these analyses are specified on the source code level, meaning that the information propagates according to the structure of the source code, and the information is usually specified in terms of source code constructions such as variables, arrays, pointers etc. This is of course convenient, since the source code is the interface between the programmer and the computer, and any information from a static analysis should be understood from the source code point of view.

However, according to the recently popularised term WYSINWYX (What You See Is Not What You eXecute) [5], there are many fallacies involved in trusting a source code level analysis. The term is referring to the fact that what is actually being executed on a machine, the *machine code*, may behave significantly different from the source code due to compiler optimisations or target platform properties. Thus, especially in safety critical software, an analysis made on the source code may be misleading. Optimisations such as removal of dead code, loop unrolling or parallelisation also affects the flow of the program.

Moreover, the executable may contain information which simply is not present in source code form. This includes which values reside in registers, and information about contents of the cache or pipeline. This kind of information is very important in analyses which extract timing behaviour from programs [34].

On source level, a program may also lack full semantic meaning. For example, a computation leading to an overflow have an undefined behaviour at the (target independent) source level. On the other hand, executable code always have exact semantics for every instruction. Thus, code where undefined behaviour is not present allows a more precise analysis.

To summarise, recently there have been an increasing interest in analysing executables rather than source code. However, as revealed in the next section, this involves a lot of difficulties.

3 Static Analysis on Executables

As hinted in the previous section, many applications benefit from performing analysis on executable code rather than source code. Though this seems simple enough at first sight, data-flow analysis in general relies on many assumptions which are valid only on the source level.

This survey investigates the challenges associated with static analysis on executable code. We will focus on some of the distinct issues that are not a problem on the source level, but which become problematic when analysis is on executable code where many assumptions commonly used in static analysis are no longer valid. We conclude this section by briefly summarising the content of the survey.

Control Flow Most static analyses rely on a graph representation of the program flow. In other words, the control flow is assumed to be (mostly) statically known. However, in executable code, the program flow typically cannot be extracted precisely. This is due to dynamic control flow such as indirect jumps. In Section 5 we will look at various techniques to perform analysis where dynamic control flow is present.

Memory Model The example analysis derived lower and upper bounds on program variables. However, in executable code, values may reside in registers, or be loaded from various memory locations. Moreover, addresses might be computed using arithmetic, and the concept of "a variable" is in general vague or not present. This is a problem, especially for analyses which tries to maintain relations between variables. Section 6 will investigate different methodologies for how to model program variables in low-level code.

Integer Representation It is often convenient to formulate data-flow analyses mathematically and assume that integer-valued variables assume values from the set of mathematical (unbounded) integers. However, in practice, at least on executable code, integers are stored as binary strings of a fixed length. Because of this, integers are bounded and effects like wraparounds are prevalent. Moreover, binary strings can often be interpreted as signed or unsigned. This is often ignored or assumed not to be a problem in source level analysis. However, many studies have indicated the importance of accurate modelling of integers. This will be covered in Section 7.

Bit-Precise Modelling Subtle bugs even on source level can be due to what happens on the bit-level in compiled code. For verification, it has lately been more common to apply SAT/SMT solvers to verify bit-level propositions. In Section 8 we will see how these tools can be used as applications in static analysis.

The rest of the paper is outlined as follows. Section 4 gives the techincal background needed to understand the paper. Section 5-8 surveys each of the problem areas stated above and Section 9 concludes the paper by a discussion.

4 Abstract Interpretation and Background

Most analyses referred to in this survey are based on *abstract interpretation* [9], which is a general framework for abstracting program semantics. One of the benefits with abstract interpretation is that it provides a systematic way of specifying data-flow analyses. The idea is that any analysis is specified as an abstraction of a programs *collecting semantics*. The collecting semantics contains a program's reachable semantic states. Thus, the collecting semantics theoretically contains exact information about which states a program can possibly

reach in any execution. This unfortunately means that the collecting semantics is in general not computable, but it is possible to soundly approximate it via *abstract states*.

An abstract state is used as a sound approximation of a set of concrete states. It is a compact representation of a set of concrete states. The representation is not expressive enough to express all possible sets of states, forcing the representation to approximate a set of states by a superset of states. Such a representation model is called an *abstract domain*, and the abstract domain restricts the actual set of states that can be represented. Table 1 summarises the most commonly used abstract domains used in literature. In fact, most of the surveyed analyses in this paper uses at least one of these domains.

Abstract domains can be divided into two classes. On one hand are the *independent domains*, which abstracts each program variable independently of one another. On the other hand there are the *relational domains* which abstract the whole set of states into one compact representation, preserving relations between the variables. Relational domains are naturally more precise (but also more costly) than independent ones. Below, we briefly explain each of the domains in Table 1.

E: The Constant Domain The least precise (of the mentioned domains), and also cheapest domain is the constant domain. The analysis derives for each program point and each variable if it is a constant, and if it is, which one. If a variable may not be a constant at a program point, the value $\top$, meaning "no information" is derived for it.

D: The Interval Domain The interval domain generalises the constant domain by inferring lower and upper bounds on each variable. This is perhaps the most used abstract domain.

F: The Congruence Domain The congruence domain also generalises the constant domain, but orthogonally to the interval domain. The domain cannot model lower and upper bounds, but instead it models the smallest congruence (w.r.t. inclusion) class containing all values a variable might have at a program point.

B: The Affine Relations Domain The relational version of the constant domain. This domain detects all affine relations between variables.

A: The Polyhedral Domain A generalisation of the affine relations domain and the relational version of the interval domain, which contains all possible program states at each program point within the smallest possible convex polyhedron surrounding them.

C: The Affine Congruences Domain Also a generalisation of the affine relations domain, and the relational version of the congruence domain.

Another nice, and commonly used property of abstract interpretation is the concept of *product domains*. Since abstract domains are specified using the

Invariant type	Inequalities	Equalities	Congruences
Relational	A: $\sum_i a_i x_i \leq c$	B: $\sum_i a_i x_i = c$	C: $\sum_i a_i x_i \equiv c \mod n$
Independent	D: $a \leq x \leq b$	E: $x = a$	F: $x \equiv a \mod n$

Table 1: Common abstract domains. They are called: A: The polyhedral domain[10], B: The affine relation domain [21], C: The affine congruence domain [17], D: The interval domain [9], E: The constant propagation domain (classic) and F: the artihmetical congruence domain [16]

same methodology, they are easy to combine to obtain more detailed domains. The simplest version is to simply run the analysis using different domains and interpret the result as the intersection of the abstract values derived by the different analyses. This is known as the *direct product* of the domains. If the information obtained from the different analyses are used optimally to benefit from each other during analysis, this is known as the *reduced product* of the domains. A commonly used reduced product is that of intervals and congruences [14, 3, 30].

5 Control Flow

Static analysis relies on the graph representation of a program to be able to propagate analysis results through the program. On source level, the control flow can often be trivially extracted from the code structure[2]. On executable level, on the other hand, the control flow is less obvious. Jumping targets might be calculated dynamically. A very common and typical case is switch cases where a jump table is usually created and the actual jump is calculated from the switch-index.

Thus, it may not be possible to extract the exact program flow directly from the code. This section lists some approaches to analyse code in the absence of exact control flow information. For reference, a lists of problems and challenges associated with finding the program flow can be found in Section 2.1 of [33].

The most common approach to control flow is to re-build the control flow graph from scratch. This is usually done bottom-up by reading and decoding one instruction at a time and building edges where jumps are found, as in [33, 19, 22]. This is straightforward and works as long as the jumps are absolute or relative absolute. However, if the target address needs to be computed, i.e., if it is a switch-statement or a function pointer or other dynamic control flow, this is a problem that needs resolving. In general there are two approaches. The first approach is to attempt to compute the target address, which usually involves some sort of pattern matching. The second approach is to mark the jump as unknown for later processing. It is common to combine approaches as well, and in the following, some publications will appear in several sections.

[2]However, there might still be some issues like function pointers or dynamic dispatch

5.1 Pattern Matching Approaches

If the compiler and/or the architecture is known, then known patterns in the code layout can be exploited to resolve dynamic jumps. Holsti [19] uses partial evaluation to find target addresses in switch tables, thus the structure of switch-tables is exploited to resolve jumps. Theiling [33] relies on a decoder for each architecture which is assumed to be able to extract branch targets from switch-tables etc.

The problem with pattern matching is that it only works for specific compilers or platforms. However, for a precise analysis on executable level, it is hard to develop generic strategies which generalise to any architecture and compiler.

5.2 Top-Down Approaches

In a top-down approach, first all basic blocks and procedures are detected, and then edges between the basic blocks are established. If the instructions are known, this can be done by simply traversing the executable and mark jumping instructions as end of basic blocks and (known) targets as starts of basic blocks [1].

De Sutter et. al. [32] were the first to use a special node representing unknown targets. This node is made such that any unresolved branch links to this node. Similarly, this node then links to each possible target. Thus, this is a conservative and safe over-estimation of the program flow. When the graph has been fully built (including this special node), pruning techniques based on constant propagation are used to (safely) remove as many incoming and outgoing edges from this special node as possible, to make the control flow more precise. This can be seen as a top-down approach in the sense that a conservative flow is first obtained, and then later refined by subsequent analyses.

Top-Down approaches are associated with the difficulty of first finding the set of instructions belonging to the program. This can be a challenge, especially when instruction sizes are not fixed.

5.3 Bottom-Up Approaches

In a top-down approach, a conservative program flow is first built and later refined. In a bottom-up approach the graph is built by traversing instruction by instruction, building blocks and edges as they become available.

Theiling [33] propose a bottom-up approach by analysing instruction by instruction in a procedure, while recording calls and recording possible target addresses. The recording of target addresses relies on a platform specific *decoder* which may also be aware of compiler structures such as branch tables (see Section 5.1 about pattern matching).

Holsti [19], uses a method where a flow graph is built bottom up. The flow graph can be seen as a context sensitive program graph, on which partial evaluation is performed to obtain target addresses.

A method entirely based on abstract interpretation is suggested by Kinder [22]. In the method, abstract interpretation (over an arbitrary abstract domain) is performed, but instead of being performed over the program graph, the propagation of values is resolved at analysis time. In other words, values that corresponds to addresses are also modelled by the abstract interpretation, and the analysis proceeds to addresses described by the abstract values. The key idea is here to model the possible target adresses in the used abstract domain. This leads to a over-estimation of addresses, but is shown to be as precise as the abstract domain is. To summarise the approach, the program graph is built during the actual analysis. A similar approach is used in [15].

5.4 Other Approaches

In [3], indirect jumps are added as edges in their graph representation of the program in two cases: if all targets can be read from program memory, or if the address can be determined by a tight value from their value set analysis (See Section 6.2). If not, the node does not add extra edges and marks the jump as an "unsafe instruction" and reports this problem to the user.

A slicing technique is used to find addresses in jumping tables in [8].

6 Memory Model

In high level languages, the memory is modelled as an environment: a map from variable names to values. In executable code, however, this modelling is not feasible. Typically the memory has to be modelled more specifically. First of all, it is no longer obvious what a program variable is. A program variable, from the high level point of view, may be a register, a memory location or an offset to the stack pointer. While this information can sometimes be obtained by debugging information from the compiler, it is not always available. This section surveys different approaches to model memory for executables in absence of debugging information.

6.1 Direct Memory Modelling

In [6], the ATMEL ATmega16 microcontroller is the platform for the analysis. This microcontroller has 1120 bytes of addressable memory, and thus, the authors simply models the memory as a mapping from a concrete address to an abstract value. The benefits of this is that since memory is directly modelled, it is easy to detect when memory areas are overwritten as well as keep track of pointers. However, such an approach is only feasible when the amount of addressable memory is small. A similar approach is used in [31] [3].

[3]However, a second layer mapping variables to memory addresses is used, so program variables still has to be identified

6.2 Value and Alias Analysis

Debray et. al. [12, 13] introduces an abstract interpretation on executables based on the congruence domain to derive possible aliasing issues. The analysis analyses the contents of the registers at each program point to discover a congruence class. Two registers are assumed to possibly alias if the congruence classes are not disjoint.

Balakrishnan and Reps introduced an analysis called value set analysis (VSA) in [3], later enhanced in [4, 29]. Here, memory is modelled as a set of memory regions, which are assumed to be non-overlapping address spaces. In particular, each procedure is associated with its own memory region containing local variables and parameters. Moreover, global variables reside in their own memory region and each dynamically allocated object gets its own memory region. One of the reasons for dividing each procedure into distinct memory regions is that the same area of the memory might be used for different things at different points in the execution (e.g., the same memory space might be used for different procedure calls).

Concrete memory locations are referred to as offsets within a memory region. Before analysis, a disassembly toolkit called IDAPro [20] is used to find all statically known addresses of the program to be analysed. Each statically known address is then associated with an abstract location, called an a-loc by the authors.

Abstract stores are then used to represent the analysis values. An abstract store maps each a-loc to a mapping from memory regions to an abstract value, or more formally:

$$\text{a-store} \ : \ \text{a-loc} \ \rightarrow (\text{memreg} \rightarrow \text{absval})$$

where absval is an abstract value from the reduced product of the integer and the congruence domain. These abstract values represents a set of possible offsets in the memory region. In this way, each a-loc can compactly be associated with a set of possible addresses.

In addition to the analysis with the product domain to find possible addresses, a relational analysis among the registers is performed. As the analysis is specified for x86 architectures, the analysis captures the relation among eight registers. Since eight is a low and constant number, this is with a constant complexity. For the relational analysis, the algorithm in [25] is used.

To summarise, the result of the analysis is that each statically known address and register (that is, each a-loc) is associated with a set of possible addresses to which it can refer.

6.3 Discovering Program Variables

In [3], IDAPro is used to identify program variables. It uses the fact that IDAPro can find all statically known addresses and uses them to identify variables (or a-locs). Global variables are identified as absolute addresses and local variables are identified as an offset from the stack pointer. So a local variable x of function

foo would be identified by the offset [foo − 16] for instance. In this way, local and global variables are discovered. However, this method has problems with dynamically allocated space and aggregates such as arrays and structs.

Balakrishnan and Reps enhance their work to handle aggregates better in [4]. The problems of the previous approach was that it typically would not recognise individual elements in structs or individual array elements. Here, VSA analysis and Aggregate Structure Identification (ASI) [28] is used to discover variables.

ASI analyses how memory is accessed from a description of data access from the program, and gives as outputs directed acyclic graphs (DAGs) describing how memory is accessed. The DAGs has two types of nodes, struct nodes and array nodes, with sub-DAGs describing their structure. For instance, ASI can detect that a sequence of bytes is used as "An array of four elements, where each element is a struct with two fields containing four bytes each".

Using aggregate information, the a-locs used in the VSA analysis can be refined. Thus, a new set of a-locs which is more representative of how variables are used is obtained from the ASI. Of course, using this new set of variables can give more precise results in the VSA analysis, and consequently, a subsequent VSA analysis is performed. However, this new VSA analysis can then give more detailed flow information, which when fed to ASI analysis can give more detailed memory access information. Thus, a fixed point analysis is performed when the output of VSA is given as input to ASI and vice versa until a stable answer is obtained.

7 Integer Representation

As mentioned in the introduction, integers are in real machines usually represented by a fixed sized binary string, while most analyses assumes that values are unbounded, resulting in possibly unsound results. This problem is mostly prevalent in abstract interpretation with the purpose of identifying possible values for variables at a program point. Most commonly used abstract domains such as [9, 16, 17, 21, 10] where originally presented with the assumption that integers are unbounded.

When a calculation results in a result which cannot be represented by a binary string of the preferred size, an *overflow* occurs. Note that different platforms may have different semantics for overflows. The most common way to handle it is to truncate the result to the desired number of bits, leading to a wrap-around effect. Since this is the most common semantics, we assume from now on that the semantics of an overflow is wrap-around.

Another problem is that the bit-strings representing integers can be interpreted as signed or unsigned. On the source level, casts between these two representations can be made. While such a cast does not affect the bit-string, it can largely affect the values an analysis has derived. For example, during a cast from signed to unsigned, a value might turn from a small negative number to a large positive one.

This section will survey various methods to adapt numeric abstract domains

to make them sound when seen as integers represented by bit-strings. There seems to be two main approaches to the problem: to model the abstract domain in modular arithmetic, or splitting abstract values into several for processing.

7.1 Modular Arithmetic

One mathematically convenient approach to solve both the problem of finite sized integers and the signed/unsigned problem, is to use modular arithmetic. That is, instead of considering integer valued variables to assume values in $\mathbb{Z}$, variables assume values from the set of congruence classes modulo n: $\mathbb{Z}_n = \{c_0, c_1, ..., c_{n-1}\}$. That is to say, the element $c_k \in \mathbb{Z}_n$ represents the class $\{k + in | i \in \mathbb{Z}\}$. The class c_k is an equivalence class, and we say the elements of c_k are *congruent modulo* n, or equivalently, let $x_0, x_1 \in c_k$, then $x_0 \equiv x_1 \mod n$. The class $\mathbb{Z}_n$ forms a ring of integers and thus have well defined addition and multiplication operators.

If integers are represented internally by binary strings of length N, then consider the set of congruence classes modulo 2^N, or $\mathbb{Z}_{2^N}$. This class contains exactly the 2^N elements representable by N bits. Moreover, modulo addition and multiplication now works exactly as they would do for binary strings, including wrap-around effects[4].

Similarly, this approach can be used to handle the signed/unsigned problems by observing that the signed and unsigned representation of any bit-string B of length N will always be congruent modulo N. However, this observation alone does not solve all the issues concerning signed and unsigned integers.

7.1.1 Affine Relations

Müller-Olm and Seidl presented an analysis of affine relations in [26]. The analysis derives all present affine equality relations among variables at each program point (if conditionals are ignored). Their approach to variables is that they assume values in the ring $\mathbb{Z}_{2^N}$, in order to make the analysis sound for values modelled as N-bit strings. They develop their method mathematically, and it can be seen as a variant of Granger's affine congruence domain, but with a fixed modulus 2^N. The paper includes algorithms for interprocedural analysis using this domain.

7.2 Splitting Abstract Environments

Another way to deal with over-flows caused by finite representation of integers or the problem with signed or unsigned integers is to, at some point, divide the abstract values of the domain. For the overflow problem, an abstract value can be divided into two parts; one part for the non-over flowing result and one

[4]If the case of an overflow, extra bits would be needed to represent a larger number, but these are simply ignored, with possibly a status flag is set. This causes the wrap-around as simulated by the modular arithmetic.

for the wrapped result. These two can then be manipulated independently and finally merged together for a safe result.

To solve the signed/unsigned problem a similar approach can be used; having one representation for the signed result and one for the unsigned. However, these results should not be merged, but rather they should be seen as safe approximations of the different interpretations. This section surveys some work when these approaches have been used.

7.2.1 Non-relational domains

In [30], a modification of the reduced product of the interval and the congruence domain was presented. The domain is specially designed for being able to handle overflows in arithmetic. The approach is to use the domain as normal, but verify for each operation if an overflow occurs. This is done by double the amount of needed bits in each operation, that is, if n bits are used to represent integers, $2n$ bits are used in the operation. They then check if n bits are enough to store the result and if not, the abstract value (i.e., the invariant) is separated into two disjoint sets, one which contains the overflow only (represented only by the lower n bits), and the rest is the non-overflowing part. The union of the two is then used as final result.

In this method, it is also possible to give invariants on the form where the lower bound is higher than the upper bound. This happens when the boundaries of a variable are close to a wrap around [5]. The authors again handle this case by separating the abstract values into two disjoint set, perform the operations point-wise, and merge the result.

Finally, the authors incorporates an affine analysis to detect linear relationships between variables to further enhance the final results.

In [18], the interval domain is used. To handle the problem of signed and unsigned integers, a product domain of signed and unsigned intervals is used. In addition, to handle overflows, an interval can be split into two intervals, one with the least value as lower bound and one with the greatest value as upper bound. Due to this dual representation, wrap-arounds can be handled.

7.2.2 Relational Domains

Simon and King have attempted to solve the problem of wraparounds and interpretation of signed and unsigned integers for the polyhedral domain [31]. Their approach does not change the polyhedral domain or the analysis it self, however, it changes the interpretation of the abstract values. First, the interpretation of the values inside a polyhedron is restricted to the values possible to represent by a bit string. Moreover, all values that can possibly be interpreted by the bit-strings are reported. In this sense, wrap-arounds are implicitly modelled by the polyhedron; any value in the polyhedron which is smaller or larger than can

[5] Concretely, if 8 bits are used, an abstraction on the form $254 \leq x \leq 2$ would mean that x would lie either in the interval $254 - 255$ or in the interval $0 - 2$.

be represented by n bits, is simply truncated to n bits, just as it is done in the concrete execution.

However, this re-interpretation of values in a polyhedron has a consequence: guards can no longer be modelled as they used to. An affine guard (or conditional) is usually handled by taking the intersection of the polyhedron with the half-space representing the guard. However, since the concrete values represented by a polyhedron might be "wrapped" values (i.e., smaller or larger than the ones present in the polyhedron), the guard might affect values not normally affected by the polyhedron. This is where the splitting of the abstract environment comes in.

Let $\mathbb{Z}_{2^n}^m$ denote the subspace of $\mathbb{Z}^m$ where each point can be represented by m binary strings of length n. If the polyhedron has points outside this subspace, the other parts are partitioned into spaces the same size as $\mathbb{Z}_{2^n}^m$, and then translated into the same base as $\mathbb{Z}_{2^n}^m$. This has the same effect as truncating the values to n bits. The guard is then applied to each of the translated subspaces, and finally the results are merged together into one polyhedron.

8 Bit-Precise Modelling

Since executable code typically involves bit-manipulation, it can often be beneficial for a program analysis to verify conditions on bit-level using SAT or SMT-solvers.

An SAT-solver (SATisfiability) is an automatic procedure for checking the satisfiability of arbitrary Boolean formulae. That is, given a Boolean formula and a set of input variables, the SAT-solver checks whether there is a satisfying assignment of the input variables such that the formula is true.

An SMT-solver (Satisfiability Modulo Theories) is a generalisation of a SAT solver in that it allows for binary-valued functions. These functions may have non-binary arguments (like integers), and may be left uninterpreted. However, theories might be used to interpret the binary-valued functions. Examples are real arithmetic, binary arithmetic, list arithmetics. The core of an SMT-solver is usually a SAT-solver, but with aid for other types of procedures solving the theories.

8.1 Bit-Level Abstract Domains

In [23], a domain able to find congruence relations between individual bits is presented. They describe how Boolean functions can be abstracted as systems of congruence constraints. Any Boolean function can be abstracted in this way by their algorithm, which involves repeated calls to a SAT-solver to incrementally build the abstraction.

The abstract domain used is exactly that of Granger [17] (see Figure 1), however, instead of modelling program variables, the individual bits of program variables are modelled. So if x, y are unsigned, byte sized program variables and $x = x_7x_6...x_0, y = y_7y_6...y_0$ are their respective bit representations, then their

analysis derives invariants of the form

$$\sum_{i=0}^{7} 2^i(a_i x_i + b_i y_i) \equiv c \mod n$$

where $a_i, b_i \in \mathbb{Z}$. This enables detailed relation between individual bits to be modelled.

Brauer et. al. also uses a bit-level domain in [6], where the abstract values are strings of generalised bits. A generalised bit is a value from the set $\{\bot, 0, 1, \top\}$ [6]. The reduced product of this domain and the interval domain is then used to obtain analysis values which take bit-strings into account.

8.2 General Static Analysis

In [24], a method to find software bugs, focusing on detection of over-flows is presented. The main contribution here is that they combine a static analysis tool called PREfix [7] with an SMT-solver Z3[11], to be able to detect errors caused by overflows. While doing this, care is taken not to report every overflow, since overflows might be fully intentional, but the one resulting in possible erroneous behaviour of the program. PREfix is based on symbolic execution, that is, execution over abstract states. Z3 is then used to check, in each computation that may result in an overflow, the satisfiability of a formula expressing the possibility of an overflow. The authors describe these formulae for all common arithmetic operators and casting operators.

9 Discussion

Static analysis and data-flow analysis in particular have been around for a long time, and many different analyses for many different purposes have been developed. However, most of these analyses have been developed in the comfortable assumptions that the program flow is known, that memory can be modelled as mappings from variables to values and that program variables take values from infinite mathematical sets. These assumptions however, are generally not feasible to use when analysing executables, and as such, analysis of executables is almost a different topic.

While analysis on source code level can be useful, the WYSINWYX phenomena states that in safety critical systems, it might not be the ideal way to go. However, to fully take account for the problems arising in executable code is difficult. Many problems severely hamper the usefulness of some proposed analyses. The problems of wrap-arounds is a difficult one, making almost all proposed numeric domains unsound. This is specially difficult for relational domains, and not too many attempts have been made solve it (with the notable exceptions of [26] and [31]).

[6] Another notation is used by the authors

What would be interesting to see is a full complete framework for analysis on executables addressing all the problems outlined in this paper: the memory model, control flow, finite sized integers and bit-preciseness. The most serious attempts to develop such a framework would be the works of Balakrishnan and Reps (see Section 6.2) and the tool Codesurfer/x86 [2], where a model for program flow, the memory model, including modelling of "program variables" is taken into consideration. Another, more general attempt to adapt abstract interpretation in general to executables is the work in [22] (Section 5.3), which includes a theoretical adaptation of the abstract interpretation in the absence of known program flow.

We believe that, since much of the work in this field is quite recent, an increased effort in adapting static analysis to executable code is happening, especially since a lot of issues still remains. None of the challenges of analysis of executables can yet be regarded as solved.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Executables Gogul Balakrishnan, Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86. In *of Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

[3] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004.

[4] Gogul Balakrishnan and Thomas W. Reps. Divine: Discovering variables in executables. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2007.

[5] Gogul Balakrishnan and Thomas W. Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6), 2010.

[6] Jorg Brauer, Thomas Noll, and Bastian Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *Proceedings of the 13th International Workshop on Software, Compilers for Embedded Systems*, SCOPES '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[8] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *In Int. Conf. on Softw. Maint*, pages 228–237. IEEE-CS Press, 1998.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[10] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[11] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[12] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. Technical report, Tucson, AZ, USA, 1997.

[13] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *POPL*, pages 12–24, 1998.

[14] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Seventh International Workshop on Worst-Case Execution Time Analysis, (WCET'2007)*, July 2007.

[15] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 188–203, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Philippe Granger. Static analysis of arithmetical congruences. In *International Journal of Computer Mathematics, Volume 30*, pages 165–190, 1989.

[17] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[18] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system c programs. In *The 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS05)*, February 2005.

[19] Niklas Holsti. Analysing switch-case tables by partial evaluation. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[20] IDAPro. The idapro debugger and disassembler. http://www.hex-rays.com/idapro/, mar 2011.

[21] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[22] Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universitat Darmstadt, 2010.

[23] Andy King and Harald Sondergaard. Automatic abstraction for congruences. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 197–213. Springer Berlin / Heidelberg, 2010.

[24] Yannick Moy, Nikolaj Bjorner, and Dave Sielaff. Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Technical report, Microsoft Research, 2009.

[25] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. *SIGPLAN Not.*, 39:330–341, January 2004.

[26] Markus Muller-Olm and Helmut Seidl. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[27] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[28] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 119–132, New York, NY, USA, 1999. ACM.

[29] Thomas W. Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *CC*, pages 16–35, 2008.

[30] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Axel Simon and Andy King. Taming the wrapping of integer arithmetic. In *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74061-2_8.

[32] Bjorn De Sutter, Bruno De Bus, Koenraad De Bosschere, P. Keyngnaert, and Bart Demoen. On the static analysis of indirect control transfers in binaries. In Hamid R. Arabnia, editor, *PDPTA*. CSREA Press, 2000.

[33] Henrik Theiling. Extracting safe and precise control flow from binaries. In *In Proc. 7th conference on real-time computing systems and applications*, 2000.

[34] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.